МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

отчет

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: БДП с рандомизацией.

Студент гр.8304 Самакаев Д.И.

Преподаватель Фирсов М.А.

Санкт-Петербург

2019

Введение

Вариант 10.

Целью данной курсовой работы является реализация БДП и демонстрация работы алгоритма. Алгоритм использует рекурсивные методы. Двоичное дерево поиска (англ. binary search tree, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X

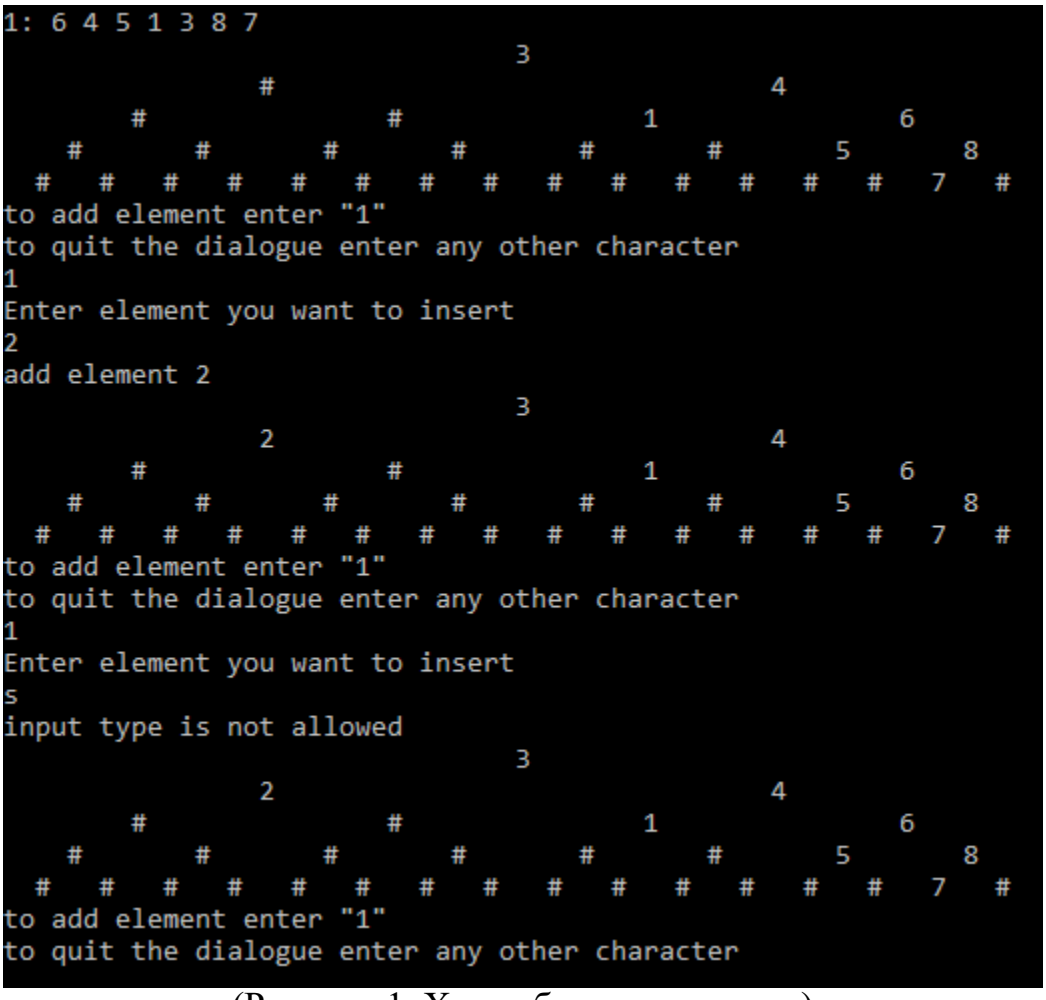
Ход работы.

Двоичное дерево поиска - это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева - левое и правое - являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

На рисунке 1 продемонстрирована работа программы.

Было реализовано общение с пользователем посредством консольного интерфейса.



(Рисунок 1. Ход работы программы)

Из тестового файла вводится строка с входными данными. На их основе строится дерево. Для этого используется рекурсивная функция заполнения БДП.

При каждом добавлении в дерево очередного элемента, происходит рандомизация (с вероятностью 1/n, где n – количество узлов в дереве, элемент запишется в голову дерева, а старое дерево станет левым или правым поддеревом).

В поле структуры, предназначенное для хранения элемента записывается значение элемента (в случае, если это поле еще не заполнено), затем эта же функция применяется для левого или правого поддеревьев (в зависимости от результата сравнения вставляемого элемента с текущим).

ФУНКЦИИ И СТРУКТУРЫ ДАННЫХ

1 Структуры:

```
template<typename T>
struct Branch {
  Branch() = default;

  std::shared_ptr<T> root = nullptr;
  std::shared_ptr<Branch> left = nullptr;
  std::shared_ptr<Branch> right = nullptr;

};
```

Структура Branch представляет узел дерева, где left –левая ветка, right – правая, а root – указатель на сам элемент.

```
template<typename T>
class BinTree {
public:
	BinTree() {
		head = std::make_shared<Branch<T>>();
	}

	bool unit_insist(T unit, std::shared_ptr<Branch<T>> temp) {
		if (!temp->root) {
			return 0;
		}

		if (*(temp->root) > unit) {
			if (!temp->left)
				return 0;
			return unit_insist(unit, temp->left);
		}

		else if (*(temp->root) < unit) {
			if (!temp->right)
				return 0;
			return unit_insist(unit, temp->right);
		}

		return 1;
	}
```

```
void update_height(std::shared_ptr<Branch<T>> temp, size_t cur_height) {

	height = std::max(height, cur_height);

	if (temp->left)
		update_height(temp->left, cur_height + 1);
	if(temp->right)
		update_height(temp->right, cur_height + 1);
}

//returns TRUE if tree already has unit and FALSE if it has not
bool Random_add_root(T unit, std::shared_ptr<Branch<T>> temp) {

	if (!unit_insist(unit, head)) {
		size_t random_tree_number = rand() % (size + 2);

		if (random_tree_number == size + 1) {

			if (!temp->root) {
				temp->root = std::make_shared<T>();
				*(temp->root) = unit;
				size++;
				return false;
			}

			if (*(temp->root) > unit) {
				std::shared_ptr<Branch<T>> new_head;

				new_head = std::make_shared<Branch<T>>();
				new_head->right = head;
				new_head->root = std::make_shared<T>();
				*(new_head->root) = unit;

				head = new_head;
				size++;
			}
			else {
				std::shared_ptr<Branch<T>> new_head;

				new_head = std::make_shared<Branch<T>>();
				new_head->left = head;
				new_head->root = std::make_shared<T>();
				*(new_head->root) = unit;

				head = new_head;
```

```
                        size++;
                    }
                }
                else addRoot(unit, temp);

                return true;
            }
            else
                return false;

}

bool addRoot(T unit, std::shared_ptr<Branch<T>> temp) {

            if (!temp->root) {
                temp->root = std::make_shared<T>();
                *(temp->root) = unit;
                size++;
                return false;
            }

            if (*(temp->root) > unit) {
                if (!temp->left)
                    temp->left = std::make_shared<Branch<T>>();
                return addRoot(unit, temp->left);
            }

            else if (*(temp->root) < unit) {
                if (!temp->right)
                    temp->right = std::make_shared<Branch<T>>();
                return addRoot(unit, temp->right);
            }

            return true;
}

void fill_tree(std::vector<T> units) {

            for (size_t i = 0; i < units.size(); i++)
                Random_add_root(units.at(i), head);

            update_height(head, 1);
}

void task(T unit) {
```

```
        if (Random_add_root(unit, head))
                std::cout << "add element " << unit << std::endl;
        else std::cout << "stucture already has element "<< unit << std::endl;

        update_height(head, 1);
}

void printTree()
{
        image.clear();

        std::string line(std::pow(2, height + 1), ' ');

        std::stack<std::shared_ptr<Branch<T> > > st;
        st.push(head);
        size_t cur_height = 0;

        while (cur_height != height)
        {
                std::vector<std::shared_ptr<Branch<T> > > nodes;
                size_t step = std::pow(2, height - cur_height + 1);
                size_t ind = std::pow(2, height - cur_height);

                while (!st.empty())
                {
                        std::shared_ptr<Branch<T>> insertElem = st.top();
                        st.pop();
                        if (typeid(T) == typeid('c') && insertElem) {
                                line.insert(ind, 1, *(insertElem->root));
                        }
                        else if (insertElem)
                                line.insert(ind, std::to_string((*(insertElem->root))));
                        else line.insert(ind, "#");
                        ind += step;

                        nodes.push_back(insertElem);
                }
                for (int i = nodes.size() - 1; i >= 0; --i)
                {
                        if (nodes[i] == nullptr)
                        {
                                st.push(nullptr);
                                st.push(nullptr);
                                continue;
                        }
```

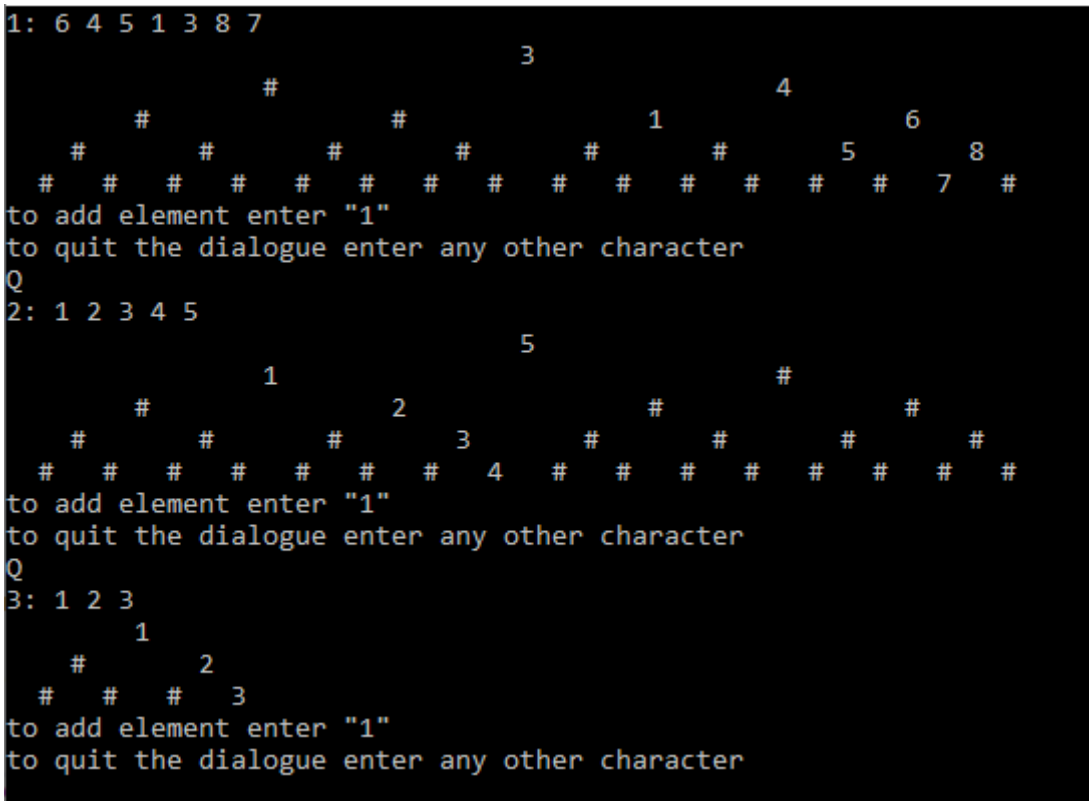
```
                        st.push(nodes[i]->right);
                        st.push(nodes[i]->left);
                }
                image.push_back(line);

                std::fill(line.begin(), line.end(), ' ');

                ++cur_height;
        }

        for (auto line : image) {
                std::cout << line << std::endl;
        }

    }

    std::shared_ptr<Branch<T>>getHead() {
        return head;
    }

private:
    std::shared_ptr<Branch<T>> head;
    size_t size = 0;
    size_t height = 0;
    std::vector<std::string> image;
};
```

Структура bin_tree используется для хранения и обработки дерева.

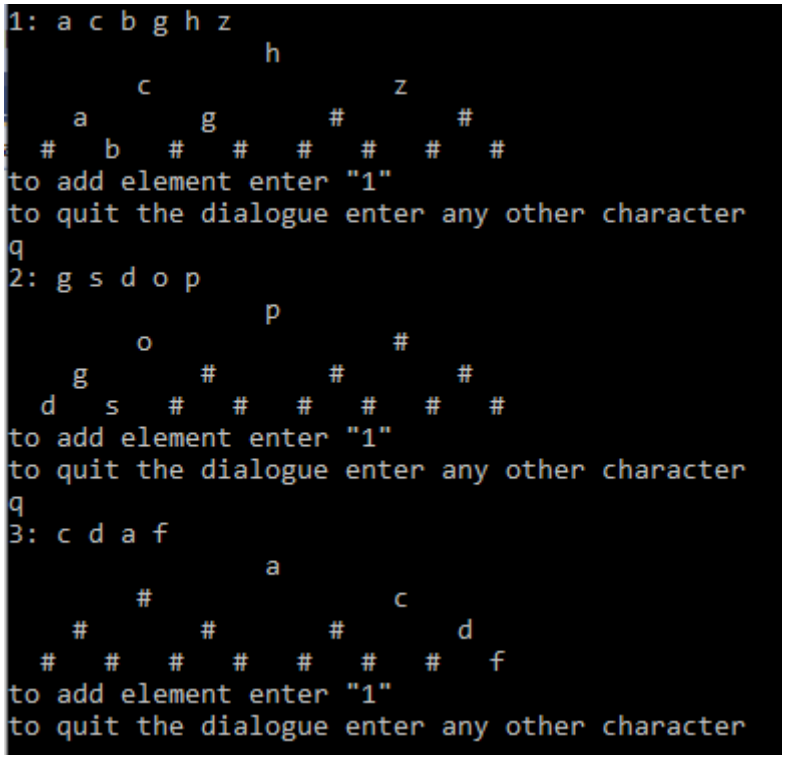
ТЕСТИРОВАНИЕ

Ниже представлены результаты работы программы для соответствующих типов данных.

int:



char:



double:

```
1: 2.2 2.12 2.35 3.21
               2.120000
        #                      2.200000
    #             #             #             2.350000
  #         #         #         #         #         #         3.210000
to add element enter "1"
to quit the dialogue enter any other character
q
2: 1.12 1.1 2.3
    1.120000
  1.100000  2.300000
to add element enter "1"
to quit the dialogue enter any other character
q
3: 0 1 2
        0.000000
    #             1.000000
  #         #         #         2.000000
to add element enter "1"
to quit the dialogue enter any other character
```

Вывод.

В ходе выполнения работы была написана программа, содержащая в себе реализацию БДП. Был получен опыт работы с дополнительными возможностями C++ и изучены алгоритмы работы с БДП. Также были закреплены знания полученные на протяжении семестра. Исходный код программы находится в приложении А.

ПРИЛОЖЕНИЕ А

СОДЕРЖИМОЕ ФАЙЛА CW.CPP

```
#include "bin_tree.h"


template<typename T>
bool str_to_t(T& item) {
	std::string str;
	std::cin >> str;

	std::stringstream linestream(str);
	linestream >> item;

	char c;
	if ((!linestream) || (linestream >> c)) {
		std::cout << "input type is not allowed" << std::endl;
		return false;
	}

	return true;
}

template<typename T>
void dialogue(BinTree<T> tree) {

	char  option;
	T  unit;

	tree.printTree();

	std::cout << "to add element enter \"1\"\nto quit the dialogue enter any other
character\n";

	option = getchar();
	getchar();
	switch (option){
	case '1':
		std::cout << "Enter element you want to insert" << std::endl;
		if(str_to_t(unit))
			tree.task(unit);
		getchar();
		dialogue(tree);
		break;
	default:
		break;
	}
}

template<typename T>
void file_input(char* argv) {

	std::ifstream file;
	std::string testfile = argv;

	file.open(testfile);
	if (!file.is_open()) {
		std::cout << "Error! File isn't open" << std::endl;
		return;
```

```
        }

        size_t i = 1;

        std::string expression;

        while (getline(file, expression)) {
                std::vector<T> units;
                std::istringstream iss(expression);

                BinTree<T> tree;
                T unit;
                while (iss >> unit) {
                        units.push_back(unit);
                }

                tree.fill_tree(units);

                std::cout << i << ": " << expression << std::endl;
                i++;

                dialogue<T>(tree);
        }

}

int main(int argc, char** argv) {

        if (argc == 1) {
                std::cout<<"Please check you entered test file name";
        }
        else file_input<int>(argv[1]);

    }
```

СОДЕРЖИМОЕ ФАЙЛА BIN_TREE.H

```
#pragma once
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <vector>
#include <sstream>
#include <cstdlib>
#include <map>
#include <stack>


template<typename T>
struct Branch {
        Branch() = default;

        std::shared_ptr<T> root = nullptr;
        std::shared_ptr<Branch> left = nullptr;
        std::shared_ptr<Branch> right = nullptr;

};

template<typename T>
class BinTree {
```

```
public:
	BinTree() {
		head = std::make_shared<Branch<T>>();
	}

	bool unit_insist(T unit, std::shared_ptr<Branch<T>> temp) {
		if (!temp->root) {
			return 0;
		}

		if (*(temp->root) > unit) {
			if (!temp->left)
				return 0;
			return unit_insist(unit, temp->left);
		}

		else if (*(temp->root) < unit) {
			if (!temp->right)
				return 0;
			return unit_insist(unit, temp->right);
		}

		return 1;
	}

	void update_height(std::shared_ptr<Branch<T>> temp, size_t cur_height) {

		height = std::max(height, cur_height);

		if (temp->left)
			update_height(temp->left, cur_height + 1);
		if(temp->right)
			update_height(temp->right, cur_height + 1);
	}

	//returns TRUE if tree already has unit and FALSE  if it has  not
	bool Random_add_root(T unit, std::shared_ptr<Branch<T>> temp) {

		if (!unit_insist(unit, head)) {
			size_t random_tree_number = rand() % (size + 2);

			if (random_tree_number == size + 1) {

				if (!temp->root)  {
					temp->root = std::make_shared<T>();
					*(temp->root) = unit;
					size++;
					return false;
				}

				if (*(temp->root) > unit) {
					std::shared_ptr<Branch<T>> new_head;

					new_head = std::make_shared<Branch<T>>();
					new_head->right = head;
					new_head->root = std::make_shared<T>();
					*(new_head->root) = unit;

					head = new_head;
					size++;
```

```
                        }
                        else {
                                std::shared_ptr<Branch<T>> new_head;

                                new_head = std::make_shared<Branch<T>>();
                                new_head->left = head;
                                new_head->root = std::make_shared<T>();
                                *(new_head->root) = unit;

                                head = new_head;
                                size++;
                        }
                }
                else addRoot(unit, temp);

                return true;
        }
        else
                return false;

}

bool addRoot(T unit, std::shared_ptr<Branch<T>> temp) {

        if (!temp->root) {
                temp->root = std::make_shared<T>();
                *(temp->root) = unit;
                size++;
                return false;
        }

        if (*(temp->root) > unit) {
                if (!temp->left)
                        temp->left = std::make_shared<Branch<T>>();
                return addRoot(unit, temp->left);
        }

        else if (*(temp->root) < unit) {
                if (!temp->right)
                        temp->right = std::make_shared<Branch<T>>();
                return addRoot(unit, temp->right);
        }

        return true;
}

void fill_tree(std::vector<T> units) {

        for (size_t i = 0; i < units.size(); i++)
                Random_add_root(units.at(i), head);

        update_height(head, 1);
}

void task(T unit) {
        if (Random_add_root(unit, head))
                std::cout << "add element " << unit << std::endl;
        else std::cout << "stucture already has element "<< unit << std::endl;

        update_height(head,  1);
```

```
	}

	void printTree()
	{
		image.clear();

		std::string line(std::pow(2, height + 1), ' ');

		std::stack<std::shared_ptr<Branch<T> > > st;
		st.push(head);
		size_t cur_height = 0;

		while (cur_height != height)
		{
			std::vector<std::shared_ptr<Branch<T> > > nodes;
			size_t step = std::pow(2, height - cur_height + 1);
			size_t ind = std::pow(2, height - cur_height);

			while (!st.empty())
			{
				std::shared_ptr<Branch<T>> insertElem = st.top();
				st.pop();
				if (typeid(T) == typeid('c') && insertElem) {
					line.insert(ind, 1, *(insertElem->root));
				}
				else if (insertElem)
					line.insert(ind, std::to_string((*(insertElem-
>root))));
				else line.insert(ind, "#");
				ind += step;

				nodes.push_back(insertElem);
			}
			for (int i = nodes.size() - 1; i >= 0; --i)
			{
				if (nodes[i] == nullptr)
				{
					st.push(nullptr);
					st.push(nullptr);
					continue;
				}
				st.push(nodes[i]->right);
				st.push(nodes[i]->left);
			}
			image.push_back(line);

			std::fill(line.begin(), line.end(), ' ');

			++cur_height;
		}

		for (auto line : image) {
			std::cout << line << std::endl;
		}

	}

	std::shared_ptr<Branch<T>>getHead() {
		return head;
	}
```

```
private:
        std::shared_ptr<Branch<T>> head;
        size_t size =  0;
        size_t height = 0;
        std::vector<std::string> image;
};
```